NFV and SDN Tutorial

Overview:

This is a basic tutorial that guides you through an experiment using OpenFlow and Network Function Virtualization (NFV). It is recommended that you first read about OpenFlow (http://archive.openflow.org/) and go over basic OpenFlow tutorial on GENI (http://groups.geni.net/geni/wiki/GENIExperiment/Tutorials/OpenFlowOVS). The goal of this tutorial is to give you hands-on experience with OpenFlow, and how it is can be used for NFV deployment. Using a basic topology, which contains two sources, a destination, two virtual network functions (IDS), an OVS and a Controller, we will show how different OpenFlow rules can be used for NFV management.

Figure 1: The image above depicts the network topology used in this tutorial.
Pre-requisites:

- A GENI account, if you don’t have one sign up!
- Familiarity with how to reserve GENI resources. (we will be using GENI Experimenter Portal as our tool).
- Familiarity with logging into GENI compute resources.
- Basic understanding of OpenFlow. OpenFlow tutorial here!
- Familiarity with the Unix Command line.
- Familiarity with the Python programming language. We will use a controller (POX Controller) written in Python for this tutorial.
- You will need to be in a project.

Tools:

- Open vSwitch.
- POX controller.

Both of the tools are already installed on the machines where the resources are provided. If it happens not to be installed, take a look at the following tutorials to learn how to install these tools:

Open Vswitch: click here for tutorial.

POX controller: click here for tutorial.

Where to get help:

- POX Wiki
- Open VSwitch
- GENI Wiki
Tutorial Instructions

Part 1: Design/Setup
• Design the experiment
• Establish the environment
• Obtain Resources

Part 2: Execute
• Login to VMs
• Configure and initialize services
• Execute Experiment

Part 3: Finish
• Tear down Experiment
• Release Resources
1. Design the Experiment

The basic topology of the tutorial is shown above. We have two sources that will communicate with the destination connected via OVS. We have two VMs (VNF1 and VNF2) running Snort Intrusion Detection Systems (IDS), as Virtual Network Functions (VNF). In this tutorial, we will put forwarding rules in OVS using controller such that whenever source wants to communicate with destination, the copy of the packets goes to one of the IDS.

Iperf will be run as an application and we will see the impact of load balancing on the performance of the network applications.

We will be using Snort IDS as an example for VNF. You can use any other Network Functions instead of IDS e.g. Firewall, Cache Proxy Server etc.
2. Establish the Environment

2.1. Pre-work:

⚠️ Skip this section if you have already established your environment for your project.

- Ensure SSH keys are setup. If your SSH keys are not setup before, do the following steps:
  - Generate an SSH Private Key on the Profile page on the GENI portal
  - Download Private Key to ~/Downloads
  - Open terminal and execute
    
    ```
    $ mv ~/Downloads/id_geni_ssh_rsa ~/.ssh/.
    $ chmod 0600 ~/.ssh/id_geni_ssh_rsa
    $ ssh-add ~/.ssh/id_geni_ssh_rsa
    ```

- Ensure you are part of a project.

3. Obtain Resources

For this tutorial, you can use resources from any aggregate in GENI. You may choose the one that pleases you better. The experiment will need following

- 1 Xen VM as OpenFlow controller with a public IP (controller)
- 1 Xen VM to be the OpenFlow switch (OVS)
- 3 Xen VMs, two Sources and one Destination (S1, S2 and destination)
- 2 Xen VMs as Virtual Network Functions (VNF1 and VNF2)

We will need two slices for this tutorial:

1. Slice for OpenFlow controller.

2. Slice for Network Topology consisting of source, destination, IDS and OVS.

To reserve resources, use GENI portal.

3.1. Controller:
- Open the slice that will run open flow controller. Reserve a VM running the controller using the RSpec that is available in the portal called XEN VM POX Ctrl.
  This is shown in the picture below.
• Chose any aggregate and reserve resources for **controller**. You will have controller VM as shown below.
3.2. Network Topology:
Create a new slice ‘Network’ for Network Topology. This topology will consist of two sources, a destination, an OVS and two VNFs. Use the rspec given below to reserve resources for ‘Network’ slice.

http://cs-people.bu.edu/nabeel/files/Network_with_NFVimage.xml

Chose any aggregate and reserve resources for Network. You will have network topology VM as shown below

![Network Topology Diagram]

3.3. Configure and Initialize:
Although OVS is installed on the host and that is meant to act as a software switch, it still needs to be configured. There are two things that need to be configured:

(1) configure your software switch with the interfaces as ports and

(2) point the switch to an OpenFlow controller.

3.3.1. Configure the Software Switch (OVS Window):

1. Login to the OVS host
2. Create an Ethernet bridge that will act as our software switch:
   ```bash
   sudo ovs-vsctl add-br br0
   ```
3. Prepare the interfaces to be added as ports to the OVS switch
   Your OVS bridge will be a Layer 2 switch and your ports do not need IP addresses. Before we remove them let’s keep some information
   - Run `ifconfig` on ovs node
   - Write down the interface names that correspond to the connections to your VNF1 and VNF2 hosts. The correspondence is
     - Interface with IP 10.10.1.13 to VNF1 - ethX
     - Interface with IP 10.10.1.14 to VNF2 – ethY

   Make sure you have noted the names of interface before you proceed. We will need the interface names to run experiments.
4. Prepare the interfaces to be added as ports to the OVS switch.
   - Your OVS bridge will be a Layer 2 switch and your ports do not need IP addresses.
     Remove the IP from your data interface.
     ```
     sudo ifconfig eth1 0
     sudo ifconfig eth2 0
     sudo ifconfig eth3 0
     sudo ifconfig eth4 0
     sudo ifconfig eth5 0
     ```

   **Be careful not to bring down eth0.** *This is the control interface, if you bring that interface down you won't be able to login to your host. For all interfaces other than eth0 and lo, remove the IP from the interfaces (your interface names may vary):*

Add all the data interfaces to your switch (bridge): Be careful not to add interface eth0. This is the control interface. The other four interfaces are your data interfaces. (Use the same interfaces as you used in the previous step).

```
sudo ovs-vsctl add-port br0 eth1
sudo ovs-vsctl add-port br0 eth2
sudo ovs-vsctl add-port br0 eth3
sudo ovs-vsctl add-port br0 eth4
sudo ovs-vsctl add-port br0 eth5
```

Now the software switch is configured! To verify the four ports configured run:

```
sudo ovs-vsctl list-ports br0
```

### 3.3.2. Point your switch to a controller

*An OpenFlow switch will not forward any packet unless instructed by a controller. Basically the forwarding table is empty, until an external controller inserts forwarding rules. The OpenFlow controller communicates with the switch over the control network and it can be anywhere in the Internet as long as it is reachable by the OVS host.*

1. Login to your controller
2. Find the control interface IP of your controller, use `ifconfig` and note down the IP address of eth0.
3. In order to point our software OpenFlow switch to the controller, in the ovs terminal window, run:
   ```
   sudo ovs-vsctl set-controller br0 tcp:<controller_ip>:6633
   ```
4. Set your switch to fail-safe-mode. For more info read the [standalone vs secure mode section](#). Run:
   ```
   sudo ovs-vsctl set-fail-mode br0 secure
   ```
5. Trust but verify. You can verify your OVS settings by issuing the following:
   ```
   sudo ovs-vsctl show
   ```
**Standalone vs Secure mode**

The OpenFlow controller is responsible for setting up all flows on the switch, which means that when the controller is not running there should be no packet switching at all. Depending on the setup of your network, such a behavior might not be desired. It might be best that when the controller is down, the switch should default back to being a learning layer 2 switch. In other circumstances however this might be undesirable. In OVS this is a tunable parameter, called fail-safe-mode which can be set to the following parameters:

- standalone [default]: in this case OVS will take responsibility for forwarding the packets if the controller fails
- secure: in this case only the controller is responsible for forwarding packets, and if the controller is down all packets are dropped.

In OVS when the parameter is not set it falls back to the standalone mode. For the purpose of this tutorial we will set the fail-safe-mode to secure, since we want to be the ones controlling the forwarding.
First thing we are doing in the experiment is pinging the VMs. By now, our switch is already configured, so we start working on the controller. As mentioned earlier, we are using POX controller and it is already installed on the controller host.

1. **Login to the hosts**
   We need to ssh to all our hosts. Again, if you don’t know how to login into a node, click [here](#) to learn.

   Open the following windows:

   - one window with ssh into the controller
   - one window with ssh into OVS
   - one window with ssh into s1
   - one window with ssh into VNF1
   - one window with ssh into VNF2
   - one window with ssh into destination

2. **Configure and Initialize services**

   **2.1. Use Learning Switch Controller**
   This is a very simple example where we are going to run a learning switch control to forward traffic from s1 to VNF1.

   1. First start a ping from s1 to VNF1, which should timeout, since there is no controller running.

      ```
      ping vnf1 -c 10
      ```

   2. The POX controller is installed under `/tmp/pox` on the controller host. POX comes with a set of example modules that you can use. One of the modules is a learning switch. Start the learning switch controller which is already available by running the following two commands:

      ```
      cd /tmp/pox
      python pox.py --verbose forwarding.l2_learning
      ```

      “l2” below uses the letter ‘l’ as in level and is not the number one. In addition, you should wait for the ”INFO ... connected” line to ensure that the switch and the controller are communicating.
The output should look like this:

```bash
nabee@controller:~$ python pox.py --verbose forwarding.12_learning
POX 0.1.0 (betta) / Copyright 2011-2013 James McCauley, et al.
DEBUG:core:POX 0.1.0 (betta) going up...
DEBUG:core:Running on CPython (2.7.6/Mar 22 2014 22:59:56)
DEBUG:core:Platform is Linux-3.12.0-33-generic-x86_64-with-Ubuntu-14.04-trusty
INFO:core:POX 0.1.0 (betta) is up.
DEBUG:openflow.of_01:Listening on 0.0.0.0:6633
INFO:openflow.of_01:[1e-d1-10-44-44-44 1] connected
DEBUG:forwarding.12_learning:Connection [1e-d1-10-44-44-44 1]
```

3. In the terminal of s1, ping VNF1:
   Now the ping should work and the output should look like this
   ```bash
   nabee@s1:~$ ping vnf1
   PING VNF1-lan2 (10.10.1.3) 56(84) bytes of data
   64 bytes from VNF1-lan2 (10.10.1.3): icmp_seq=1 ttl=64 time=40.3 ms
   64 bytes from VNF1-lan2 (10.10.1.3): icmp_seq=2 ttl=64 time=1.38 ms
   64 bytes from VNF1-lan2 (10.10.1.3): icmp_seq=3 ttl=64 time=1.34 ms
   64 bytes from VNF1-lan2 (10.10.1.3): icmp_seq=4 ttl=64 time=0.944 ms
   64 bytes from VNF1-lan2 (10.10.1.3): icmp_seq=5 ttl=64 time=1.09 ms
   ```

4. Go to your controller host and take a look at the printouts. You should see that your controller installed flows based on the mac addresses of your packets.

2.2 NFV OVS controller

Now we are going to run a different controller with openflow rules to support NFV. In this controller, the traffic shall go from source to destination, and duplicate packets are sent to IDS nodes (VNF1 or VNF2) for Intrusion detection. Picture below shows traffic between source1 and destination is red line, and the green line shows the duplicate traffic that is send to VNF1 for intrusion detection.
3. Execute Experiments:
First we need to configure the configuration files for the NFV openflow controller.

- We will first remove the default files for controller and replace them with our controller files. Execute following
  
  ```
  cd /tmp/pox/ext
  sudo chmod 777 ../ext/
  sudo rm *
  wget http://cs-people.bu.edu/nabeel/files/OVS_files.tar.gz
  tar -xvf OVS_files.tar.gz
  ```

- Now you should have different files for OVS controller. Open `port.config` file to configure the system parameters. You can use any editor to edit the file. I am using `nano` here as an example.

  ```
  nano port.config
  ```

- You will see a window as shown below. Change the value of `vnf1_interface` and `vnf2_interface` to the values that you noted down in section 3.3.1 in Design/Setup section of this tutorial. These values will tell controller the interface that is connected with VNF1 and VNF2. Example is shown below

  ![Example of port.config file]

  ```
  [general]
  destination = 10.10.1.5
  source1 = 10.10.1.1
  source2 = 10.10.1.2
  vnf1 = 10.10.1.3
  vnf2 = 10.10.1.4
  vnf1_interface = eth5
  vnf2_interface = eth1
  controller_type = RR
  file_path_pi = /tmp/pox/ext/NEV_ratio_PI.txt
  ```
Experiment 1: Load Balancing using Round Robin load balancer

In the first experiment, we will look at Round Robin load balancer for VNF Snort applications.

Snort will be running as IDS on VNF1 and VNF2 and we will try to balance the load across VNF by directing each new flow request to one of the two VNF instances in a round robin fashion.

We will use Netcat application to generate traffic between source and destination.

Netcat is a useful application for network debugging and investigation. It is used for reading from and writing to network connections using TCP or UDP. In other words, it is like talking on the phone. One talks (a node) and another one, on the other side, listens and talks back. Using netcat, you can see the communication between source and destination.

1. Open controller window and execute
   
   ```
   cd /tmp/pox/ext
   nano port.config
   ```

   Here change the controller_type to RR as shown below

   ```
   vnf2 = 16.16.1.4
   vnf1_interface = ethX
   vnf2_interface = ethY
   controller_type = RR
   ```

2. Run NFVcontroller in controller window
   
   ```
   cd /tmp/pox/
   python pox.py --verbose NFV_controller
   ```

3. Open VNF1 and VNF2 windows and run Snort IDS on each of the VNFs as shown below
   
   ```
   sudo /usr/local/bin/snort -A console -v -c / -i eth1
   ```

   You should see the window as shown below on VNF1 and VNF2 windows

   ![Snort output](image)

4. In the destination window, run netcat server
   
   ```
   nc -u -l 5000
   ```

5. In the s1 window, run netcat client to send traffic to destination node
   
   ```
   nc -u destination 5000
   ```

6. Type something on s1 window and you should see it on destination window.

   Now if you look at VNF1 and VNF2 windows running Snort IDS, you could see packets arriving at VNF1 and VNF2 node in Round Robin fashion. Example of such an output is shown below on VNF window running Snort IDS.

   ![Netcat output](image)
In the controller window, you can see that controller choses VNF1 and VNF2 is Round Robin fashion as shown below

```
('*** NFV Selected for flow: ' , 1)
DEBUG:NFV_controller:Got a packet : [02:37:31:59:60:0b:02:af:0e:c2:b0:f0:0 IF]
('*** NFV Selected for flow: ' , 2)
DEBUG:SimpleL2Learning:Installing flow for 02:af:8e:c2:b0:f0:3 -> 02:af:0e:c2:b0:f0:0 IF]
('*** NFV Selected for flow: ' , 1)
DEBUG:NFV_controller:Got a packet : [02:37:31:59:60:0b:02:af:0e:c2:b0:f0:0 IF]
('*** NFV Selected for flow: ' , 2)
```

Optional: Review OVS controller Code:

Round Robin code is based on following algorithm.

<table>
<thead>
<tr>
<th>Algorithm 3 Round Robin based OVS controller</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> Flows</td>
</tr>
<tr>
<td>1: vnfsSelected = IDS1</td>
</tr>
<tr>
<td>2: for all f in Flows do</td>
</tr>
<tr>
<td>3: if vnfsSelected == IDS1 then</td>
</tr>
<tr>
<td>4: vnfsSelected = IDS2;</td>
</tr>
<tr>
<td>5: else</td>
</tr>
<tr>
<td>6: vnfsSelected = IDS1;</td>
</tr>
<tr>
<td>7: end if</td>
</tr>
<tr>
<td>8: sendFlow(f, vnfsSelected);</td>
</tr>
<tr>
<td>9: end for</td>
</tr>
</tbody>
</table>

If you want to see the code for Round Robin load balancer, you can see it by executing

```
cd /tmp/pox/ext

```
cat NFV_controller.py
```

You can see code for Round Robin below

```python
elif self.controller_type == 'RR':
    if self.vnfs_selected == 1:
        self.vnfs_selected = 2
    elif self.vnfs_selected == 2:
        self.vnfs_selected = 1
```

Here if VNF selected for previous flow was VNF1, VNF2 is selected as VNF for next flow and vice versa. Once VNF is selected, packets are sent to the VNF by following code
if (packetSrcIp(self.packet, self.source1, log) or packetSrcIp(self.packet, self.source2, log)) and packetDstIp(self.packet, self.destination, log):
    # send according to selected VNF
    if self.vnf_selected == 1:
        self.forward_packet([out_port, self._of_duplicate_port_vnf1])
        return
    else:
        self.forward_packet([out_port, self._of_duplicate_port_vnf2])
        return
Experiment 2: Load Balancing using Proportional Integral (PI) load balancer

In this experiment, we will use control theoretic approach to do load balancing. System overview is shown in the figure 2.

In the control theoretic approach, load on the VNF1 and VNF2 is monitored, and flow-forwarding decisions are made based on the load information retrieved from the VMs.

We will run RINA distributed application to get the state of the VNFs to the controller VM. Once controller has the IDS load information, it will use the Proportional Integral (PI) load balancer to balance the load across VNFs based on the load information. This load balancing information in then further provided to the OVS controller, which updates the openflow rules on the OVS switch to balance the load.
RNA Distributed Application:

First we will run RNA distributed application to collect the VNF load information on the controller node.

We need java installed on the nodes to run RNA application. Install java on VNF1, VNF2 and controller node. To install java, execute

```
sudo apt-get install openjdk-7-jdk
```

1. In the controller window, download RNA code using following command to download RNA controller code.
```
  cd ~
  wget http://cs-people.bu.edu/nabeel/files/Control.tar.gz
  tar -xvf Control.tar.gz
```

2. Type `ifconfig` to get the IP address on the controller. Save this address as we will need this address to direct VNF RNA processes to controller RNA process.

3. In a new VNF1 window, download RNA code using following commands to get RNA VNF1 code.
```
  cd ~
  wget http://cs-people.bu.edu/nabeel/files/VNF1.tar.gz
  tar -xvf VNF1.tar.gz
```

4. In a new VNF2 window, download RNA code using following commands to get RNA VNF2 code.
```
  cd ~
  wget http://cs-people.bu.edu/nabeel/files/VNF2.tar.gz
  tar -xvf VNF2.tar.gz
```

5. Now we will change the IP address in the configuration files on VNF1, VNF2 and Controller RNA applications, so they can talk to each other.

In the VNF1 window, execute
```
  cd ~/VNF1
  nano ipcVNF1.properties
```

At the bottom of the file, change the `rina.dns.name` and `rina.idd.name` to the IP address of the controller. Following is shown as an example.

```
################################
rina.dns.name = 123.456.789.123
rina.dns.port = 58800

#RINA IDD info
rina.idd.name = 123.456.789.123
rina.idd.port = 58881
```

In the VNF2 window, execute
```
  cd ~/VNF2
  nano ipcVNF2.properties
```

At the bottom of the file, again change the `rina.dns.name` and `rina.idd.name` to the IP address of the controller.

In the controller window, execute
```
  cd ~/Control/RINA/
  nano ipcControl.properties
```

At the bottom of the file, again change the `rina.dns.name` and `rina.idd.name` to the IP address of the controller.
6. To run RINA application, do following steps.
   - In the controller window, execute following commands:
     ```
     cd ~/Control/RINA/
     ./run_controller.sh
     ```
   - In the VNF1 window, execute following commands:
     ```
     cd ~/VNF1/
     ./run_VNF1.sh
     ```
   - In the VNF2 window, execute following commands:
     ```
     cd ~/VNF2/
     ./run_VNF2.sh
     ```

   You should see output on the controller window as shown below:
   ```
   RINA application on VNF1 and VNF2 should be run as soon as possible after Controller RINA application is started. If you wait for too long, you will get null values for CPU usages, as VNF’s RINA app is not able to register with controller’s RINA app.
   ```

   If you see output on controller RINA app with null values, it means that VNF has not been able to successfully register with RINA application. You should restart all RINA processes. Sample output where VNF2 is not able to register with controller RINA app is shown below:
   ```
   To stop all RINA processes running on a VM, run `killall -v java`
   ```
PI controller:

PI-controller gets the load information of NFV1 and NFV2 using RINA distributed application and makes the load balancing decision.

The block diagram of the Proportional Integral (PI) controlled NFV system is shown in Figure 3.

![Block diagram of the PI-controller NFV system. System load L and target load T(s)=T/s of VNF1 is used to compute X, i.e. ratio of traffic diverted to VNF2. K'=K/T.](image)

Figure 3: Block diagram of the PI-controller NFV system. System load L and target load T(s)=T/s of VNF1 is used to compute X, i.e. ratio of traffic diverted to VNF2. K’ = K/T.

The RINA-based distributed monitoring application provides the VNF1 state (average CPU load) information L(t) to the PI controller. The maximum capacity of a VNF instance is T. If the load on VNF1 exceeds T, new traffic flows are forwarded to a second VNF instance, VNF2. Assuming instantaneous feedback / measured load L(t), the PI control equation is given by:

\[ x(t) = \max[0, \min[1, x(t - 1) + K\left(\frac{L(t)}{T} - 1\right)]] \]

Code for PI controller is based on following algorithm. Input IDS\text{load}.txt is the file generated by RINA distributed application. This file has load information of VNF.

**Algorithm 1 PI controller**

**Input:** IDS\text{load}.txt  
**Output:** x(t)

1. \( T = 0.5 \)
2. \( x(t - 1) = 0.0 \)
3. \( x(t) = 0.0 \)
4. \( K = 0.2 \)
5. while True do  
6. \( L(t) = \text{getLoad(IDS\text{load}.txt)}; \)  
7. \( x(t) = \max[0, \min[1, x(t - 1) + K\left(\frac{L(t)}{T} - 1\right)]]; \)  
8. \( \text{write}(t, x(t)); \)  
9. end while

1. To run PI-controller, open a new controller window and execute  
   ```bash  
   cd ~Control/PI_controller  
   python PI_controller.py ~/Control/RINA/NFV1.txt  
   ```
   Note that here we are directing PI_controller.py to the NFV1.txt file that is constantly updated by the RINA distributed application with the load information of VNFs.

2. You should see the VNF state information printed on the screen. Sample output is shown below.

```
Target CPU load at VNF1: 30.0  
Current CPU load at VNF1: 2.5280800000000005  
% of flows to be send to VNF2: 0 %  
% of flows previously send to VNF2: 0 %
```
Here target load is 30.0% of CPU usage i.e. if CPU load on VNF1 is more than 30.0%, traffic will be diverted on VNF2. Current CPU load shows the load on VNF1. Next value shows the percentage of flows that will be directed to VNF2 and last value shows the flows that were being directed to VNF2 before the current reading.
PI-based OVS controller:

Now we will run OVS controller that will get the load information from the PI-controller and direct the flows accordingly.

1. First we will update the `port.config` file to direct controller to the `NFV_ratio_PI.txt` file generated by the PI-controller, which has the load balancing information. In a new controller window, execute
   
   ```
   cd /tmp/pox/ext
   nano port.config
   ```

   Change the value of `file_path_pi` to `/tmp/pox/ext/NFV_ratio_PI.txt` and `controller_type` to `PI` as shown below.

   ```
   vnf1_interface = eth0
   vnf2_interface = eth1
   controller_type = PI
   file_path_pi = /tmp/pox/ext/NFV_ratio_PI.txt
   ```

2. Now we can run the OVS controller. Execute
   
   ```
   cd /tmp/pox
   python pox.py --verbose NFV_controller
   ```

Run Snort and Generate Traffic:

1. First we will run Snort IDS on VNF1 and VNF2. In separate windows for VNF1 and VNF2, execute following command
   
   ```
   sudo /usr/local/bin/snort -A console -v -c / -i eth1
   ```

   You should see the window as shown below on VNF1 and VNF2 windows

   ```
   * Snort: <*
   0:05.07 < Version 2.9.8.2 GRE (Build 333)
   By Martin Roesch & The Snort Team: http://www.snort.org/contact@team
   Copyright (C) 2014-2015 Cisco and/or its affiliates. All rights reserved.
   Copyright (C) 1998-2004 Sourcefire, Inc., et al.
   Using libpcap version 1.1.1
   Using ZLIB version: 1.2.3.4
   ```

2. We will use snort application to generate flows between source and destination. If iperf is not installed on your nodes, execute
   
   ```
   sudo apt-get install iperf
   ```

3. Run iperf server on destination node
   
   ```
   iperf -s
   ```

4. Now we will generate traffic from source (s1 and s2) to destination node using `iperf` and see how it effects the CPU utilization at VNF1 and VNF2 running Snort IDS.

   Note that if we run multiple instances of `iperf`, we can generate significant load on VNF instances. To run `iperf` client on source, execute
   
   ```
   iperf -c destination -t 500 &
   ```

   Note that you can run multiple instances of `iperf` by running `iperf -c destination -t 500 & multiple time in s1 and s2 nodes. This flow lasts for 500 seconds. To kill all the flows generated at a node, run `killall -v iperf`
5. Now if you look at controller window, which is running PI-controller, you can see the load on VNF1 has significantly increased. If the load is more than 30%, some percentage of traffic will be diverted to VNF2. Sample output is shown below

<table>
<thead>
<tr>
<th>Target CPU load at VNF1:</th>
<th>38.9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Current CPU load at VNF1:</td>
<td>38.123999999999995</td>
</tr>
<tr>
<td>% of flows to be send to VNF2:</td>
<td>28.6206666667%</td>
</tr>
<tr>
<td>% of flows previously send to VNF2:</td>
<td>15.2125666667%</td>
</tr>
</tbody>
</table>

Optional: Code Review: NFV Controller gets the load balancing information from the output text file generated by PI-controller. Based on value of files (variable X), it sends each new flow to either VNF1 or VNF2. Algorithm for PI-based OVS controller is shown below

```
Algorithm 2 PI-based OVS controller

Input: Flows, x(t)

1: for all f in Flows do
2:     random = generateRandom();
3:     if random > x(t) then
4:         vnfsSelected = IDS1;
5:     else
6:         vnfsSelected = IDS2;
7:     end if
8:     sendFlow(f, vnfsSelected);
9: end for
```

Code section can be found in the file /tmp/pox/ext/NFV_controller.py as shown below

```python
def _PIinvfSelection(self):
    # get the value for load balancer from PI controller output file
    f = open(self.file_path_pi, 'r')
    txt = f.read()
    txt = txt.strip(\n"
    # value is saved on 'X' variable
    X = float(txt.split('=')[1])

    # generate a uniform random number between 0 and 1
    ran = R.random()

    # if generated number is > X, then send to VNF1, else send to VNF2
    if ran > X :
        return 1
    elif ran <= X :
        return 2
    else :
        log.debug("Error reading PI-controller generated load file!")

    return -1
```
Tear down Experiment and Release Resources:

After you are done with this experiment release your resources. In the GENI Portal select the slice click on the "Delete" button. If you have used other tools to run this experiment than release resources as described in the Prerequisites for Tutorials on reservation tools pages.

Now you can start designing and running your own experiments!

Sources:

Some parts of this tutorial are taken from GENI OpenFlow Tutorial:

http://groups.geni.net/geni/wiki/GENIExperimenter/Tutorials/OpenFlowOVS

Created by: Nabeel Akhtar